Sveučilište u Zagrebu
Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroelektroniku, računalne i inteligentne sustave

Web Architecture, Protocols, and Services

Arhitektura, protokoli i usluge weba

UNIZG-FER 222464

Microservices

Mikro usluge

Klemo Vladimir

Outline

- What are microservices?
- Microservices and SOA
- Microservices modeling
- Microservices integration
- Microservices testing
- Microserviecs deployment

What are microservices?

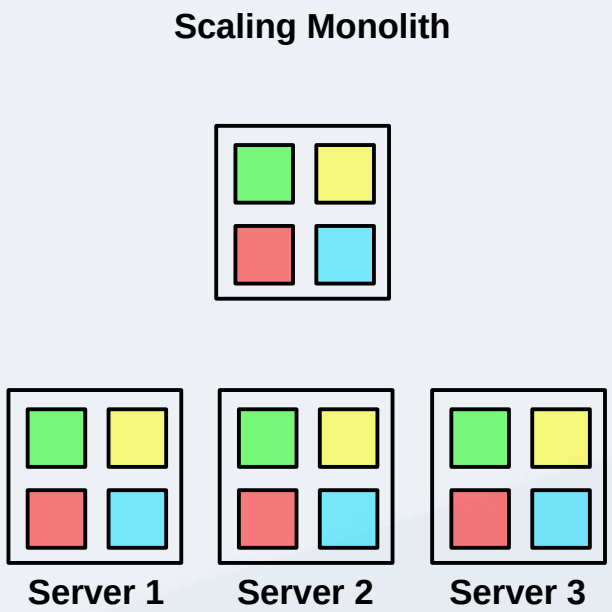
- Etymology: since **2011**
- Concept itself not new
- **Architectural style**
- Way of designing software applications as suites of **independently deployable** services
- An approach to **distributed systems** that promotes the use of **finely grained** services with their **own lifecycles**, which **collaborate** together

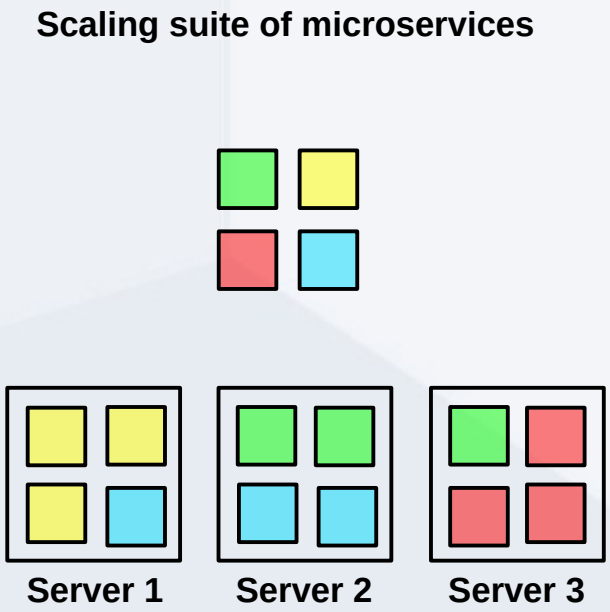
What are microservices?

- Single application as a **suite of small services**
  - build around business capabilities
  - running in its **own process**
  - **communicating** with lightweight mechanisms
    - HTTP resource API
    - RPC
  - **automatically** deployable
  - can use different programmming languages and platforms
  - can be managed by different teams

Monolithic app. vs. microservices

- *Monolithic application*
  - single program, single platform
  - any change requires rebuilding
  - scaling is hard





Microservices and SOA

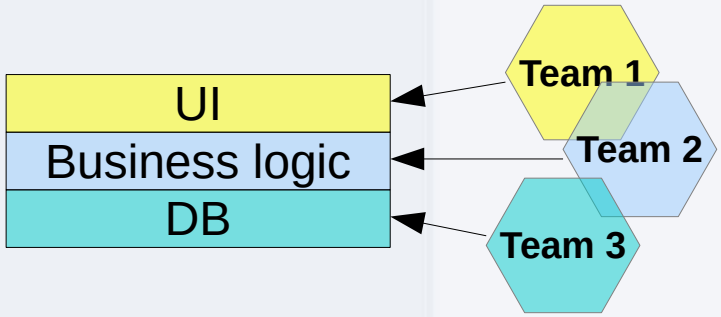
- Many similarities
  - loosely-coupled self-contained services
  - services communicate over network
  - service interoperability
  - service composition
- SOA
  - XML based (WS-*, SOAP/WSDL)
  - centralized governance model
  - focus on “enterprise”
- Microservices
  - fine grained SOA
  - web standards
  - choreography over orchestration

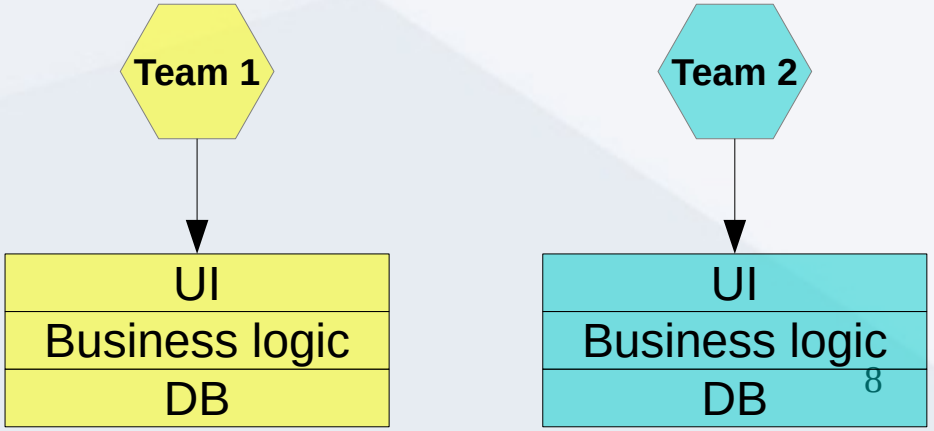
Microservices modeling

- Loose coupling
  - change to one service should not require a change to another
- High cohesion
  - group related behavior together
- Bounded context
  - specific responsibility enforced by explicit boundaries
  - internal representations that do not need to be communicated outside
  - representations that are shared externally with other bounded contexts
  - microservices should cleanly align to bounded contexts

Microservices modeling

- Business capabilities
  - Conway's law
    - "*Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure*."
  - Siloed functional teams
    - traditional approach
    - division by technology
      - DBA, business logic, UI
  - Microservices
    - division by business capabilities
    - cross-functional teams (full-stack)





Microservices modeling

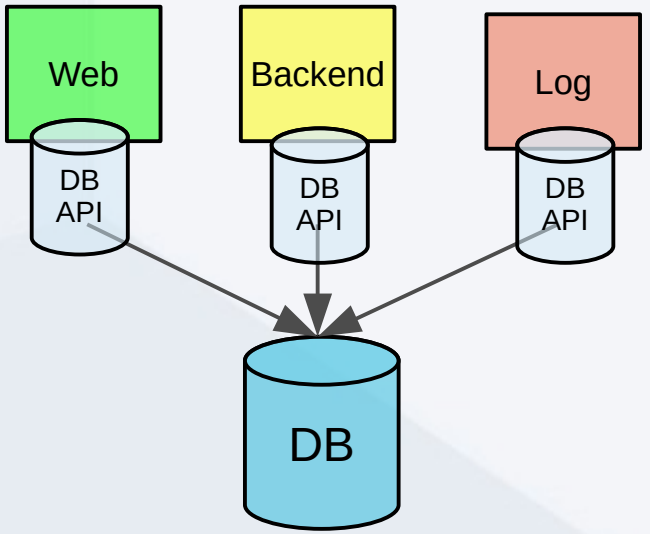
- Smart endpoints - dumb middleware
  - the smarts live in the end points (services) that are producing and consuming messages
  - microservices should **own their own domain logic**
    - receiving a request, applying logic as appropriate and producing a response
    - like filters in the classical Unix sense

Microservices integration

- Integration goals
  - avoid *breaking changes*
    - e.g. if a microservice adds new fields to data it sends out → existing consumers shouldn’t be impacted
    - multiple interfaces/versions
  - technology-agnostic APIs
  - simple APIs treated as a public interfaces
  - hide implementation details
    - avoid leaky abstractions

Microservices integration patterns

- *Shared database API*
  - **anti**-pattern
  - not technology-agnostic
  - implementation details not hidden
  - not cohesive
    - app. logic shared and distributed



Microservices integration

- Synchronous vs Asynchronous
  - Synchronous
    - call blocks until operation completes
    - easier to reason about
  - Asynchronous
    - caller doesn't wait
    - useful for long-running jobs
- Request/response vs Event-based
  - Request/resonse
    - sync/async
  - Event-based
    - asynchronous
    - highly decoupled

Microservices integration

- Request/Response: Remote Procedure Call (RPC)
  - execute remote calls as local calls
  - coupled: Java RMI
  - decoupled: SOAP, gRPC, Thrift
    - shared interface definition (WSDL, protobufs)
    - e.g. server in Java, client in Python
  - text-based
    - SOAP (XML)
  - binary
    - protobufs, Java RMI, Thrift
  - slower than local call (network)

Microservices integration

- Request/Response: REST
  - based on the Web
  - exposed representation decoupled from internal resources
  - HTTP
    - Methods like GET and POST
    - Benefit from large HTTP ecosystem
      - auth, caching, monitoring, testing …
  - Downsides
    - Client stub creation is not easy
    - Performance (compared to binary protocol)

Microservices integration

- Event-based systems
  - Message brokers
    - Producers use an API to publish an event to the broker
    - The broker informs consumers when an event arrives

Microservices integration

- Versioning
  - Pick right technology
    - shared database API vs REST
  - *Tolerant reader* pattern
    - design the client to extract only what is needed
    - ignore unknown content
    - expect variant data structures
    - example: XML/XPath

Microservices integration

- Versioning
  - Semantic versioning
    - MAJOR.MINOR.PATCH
      - MAJOR → backwards incompatible
      - MINOR → backwards compatible
      - PATCH → bug fixes to existing functionality
  - Multiple endpoints
    - V1, V2, …
      - /api/v1, /api/v2
      - request headers

Microservices integration

- Third-party integration
  - customizations on a platform you control
  - *Facade* pattern
    - Simplification
    - Reduce dependency on internals
  - *Strangler* pattern
    - Create a new system *around* the old one
    - Event interception
    - Many interceptors (microservices)

Microservices deployment

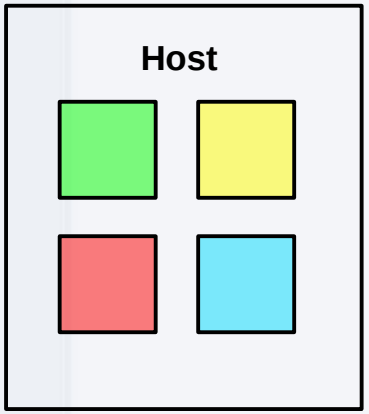
- Deployment
  - activities that make a software system available for use
- Techniques and technologies
  - Platform-specific artifacts
    - Java JAR, Python egg or Ruby Gem
    - Tools: Ansible, Chef, Puppet
  - Operating system artifacts
    - Debian/Ubuntu deb, RedHat/CentOS RPM
  - Problems
    - Artifact deployment is slow

Microservices deployment

- Custom images
  - Build image once
  - Launch copies
  - Image types
    - Dependencies only images
    - Service images (e.g. AWS AMI)
      - Image becomes artifact
  - Drawbacks
    - Image generation is slow and images are large
- Container technology
  - Docker

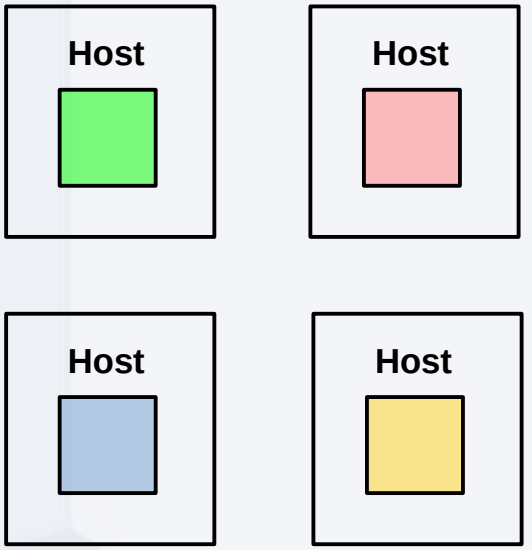
Microservices deployment

- Service-host mapping
  - Multiple services per host
    - Simpler
    - Cheaper
    - Developer-friendly
    - Problems
      - Resource sharing
        - Harder monitoring
      - Single point of failure
      - Deployment
        - Heterogeneous services on single host
        - Limited artifact options



Microservices deployment

- Service-host mapping
  - Single service per host
    - No single point of failure
    - Easier to scale
    - Alternative deployment techniques
    - Problems
      - Cost
      - Additional complexity



Microservices deployment

- Virtualization
  - Type 2 virtualization
    - KVM, VMWare, Xen, VirtualBox, …
    - Hypervisor on top of host OS
  - Vagrant
    - Deployment platform
    - Simplifies creation of production-like virtual environments on the local machine

Microservices deployment

- Virtualization alternative
  - Linux containers (LXC)
  - Container is a subtree of the overall process tree
  - No need for hypervisor
  - Kernel is shared
  - Light-weight containers
  - Fast startup (ms to s)
  - Problem: some isolation/security issues
- Hybrid
  - Containers in virtualized machines

Microservices deployment

- Docker
  - *de facto* standard deployment technology
  - Docker images
    - Application packaged with all its dependencies
    - Created with Dockerfile
  - Docker registry
    - Store and version Docker images
  - Kubernetes
    - Container orchestration

Microservices deployment

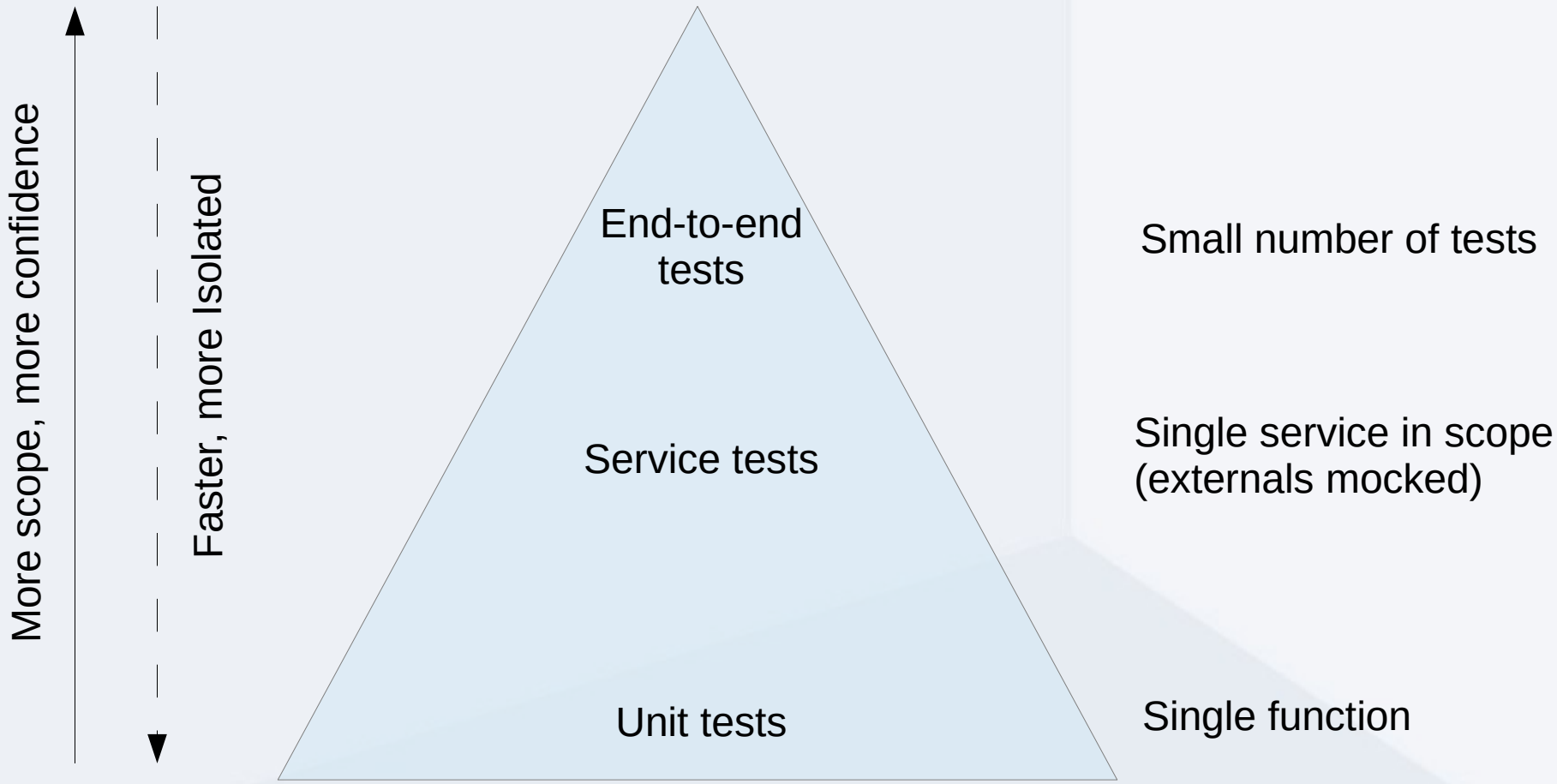
- Docker
  - Basic workflow:
    - `docker pull ubuntu`
    - `docker run ubuntu echo "hello world"`
  - Dockerfile

```
FROM ubuntu:20.04
RUN apt-get install [dependencies]
ADD app /var/apps/app
CMD ["/var/apps/app/start.sh"]
```

Microservices testing

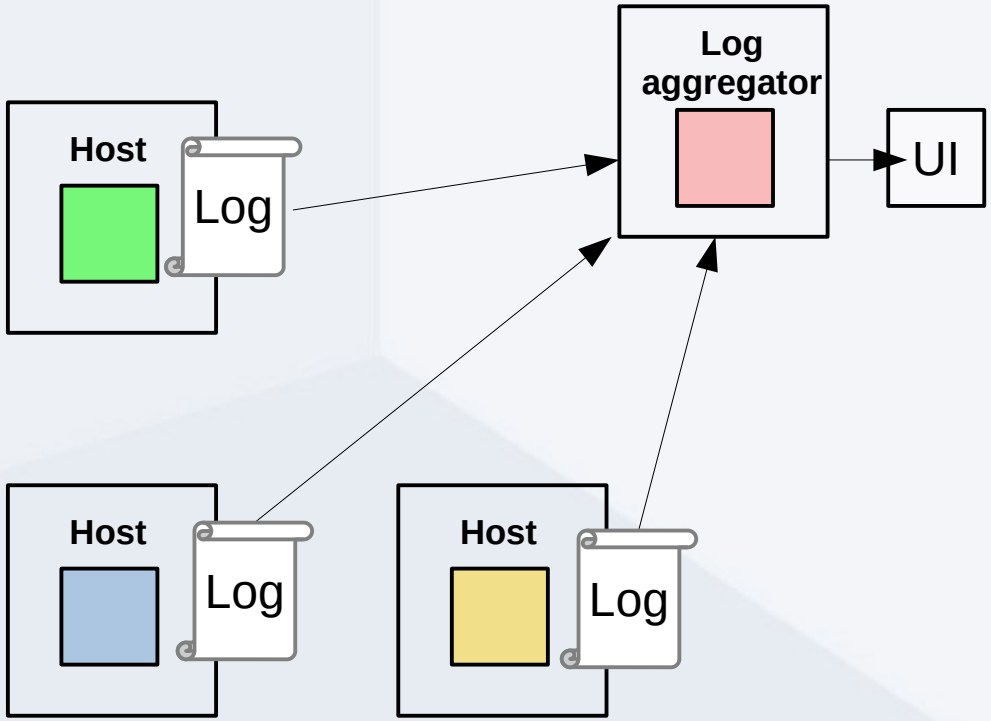
- Types of tests
  - Unit tests
    - Test a single function/method call
    - Many, fast
  - Service tests
    - Bypass UI (headless mode)
    - Test single service (fake externals)
  - End-to-End tests
    - Test entire system
    - More business facing
    - Often initiated by automated GUI actions

Microservices testing



Microservices monitoring

- Monitoring choreography of microservices
  - Collection and central aggregation of logs
  - Metrics standardization
    - Standard location, standard format
  - Correlation IDs
  - Queryable tool
  - Tools
    - Logstash
    - Nagios
    - Graphite
    - Kibana/Graphana



Conclusion

- Microservices
  - Modeled around business concepts
  - Hidden implementation details
  - Loosely coupled
  - Automated and independent deployment
  - Good
    - Scalability, componentization/modularity, fault isolation, easier deployment, legacy integration
  - Bad
    - Additional complexity, network calls, non-trivial testing and logging, support for transactions, many interfaces/protocols

Literature

- **Required reading**
  - Fowler, M., Lewis, J.
    - *Microservices, a definition of this new architectural term*. http://martinfowler.com/articles/microservices.html, 2014
- Books
  - Newman, Sam
    - *Building Microservices,* O'Reilly Media, Inc., 2015.
  - Daigneau, Robert
    - *Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services,* Addison-Wesley, 2011.